Hugo Matos de Rangel Tavares

Testing Deterministic Regular Expressions

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Janeiro de 2020

Hugo Matos de Rangel Tavares

Testing Deterministic Regular Expressions

Curricular Project Report

Supervisor: Nelma Moreira and Rogério Reis

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Janeiro de 2020

Contents

Chapter 1

Introduction

Regular expressions are widely used for matching. Thus it is important to have efficient matching algorithms of a word against a regular expression. One way would be to convert the regular expression into a deterministic finite automaton (DFA) but this automaton can be exponentially larger than the expression. In alternative, nondeterministic finite automata (NFA) equivalent to a given expression can be at most quadraticly larger.

However testing membership for a NFA is more costly. In this context, deterministic regular expressions are expressions that when simulated by Glushkov automaton the corresponding automaton is deterministic [1]. These expressions are most interesting and are in general used in applications such as XML schemas or network intrusion detection. One problem then is to know, given a general regular expression if it is deterministic. The conversion to Glushkov Automaton is quadratic in the size of the expression. Recently a linear time test was developed [3]. In this work we studied this method and implemented it using the fado.dcc.fc.up.pt, a library for manipulation of formal languages representations. This report is organized as follows.

First we describe some fundamental definitions such as regular expressions and both deterministic and non-deterministic finite automata, DFA and NFA respectively. Next we explain the construction method of Glushkov automaton and the sets that define it. Then, we present a method that checks whether a regular expression is deterministic or not that runs in $O(|e|)$, with e as the length of the tested regular expression. In the appendix we exhibit a Python 2.7 program with a complete tree structure for regular expressions, its construction and a few auxiliary functions to test determinism in a regular expression.

Chapter 2

Elementary Definitions

2.1 Regular Expressions

A regular expression, or regex, is a sequence of characters and rules that define a search pattern. A string matches a certain regex if and only if you can build the same string with the regex pattern.

Having a finite set of symbols (which we call alphabet denoted by Σ), a word is a sequence of symbols of Σ. For instance, $\Sigma = \{a,b\}$, then *aabb* is a word. The ε word is the empty sequence. The set of every words is denoted by Σ^*.

A language L is a subset of Σ^*, L $\subseteq \Sigma^*$. A regular expression over Σ defines a language and the set of regular expressions R is:

- if a $\in R$, a is a regular expression and L(a) = {a}
- $\varepsilon \in R$, $L(\varepsilon) = \{\varepsilon\}$
- e, $e' \in R$, iff e is a regular expression and e' is a regular expression, then e+e' and ee' are regular expressions in R.
- L(e+e') = L(e) $\cup$ L(e').
- L(ee') = L(e) . L(e') = { xy | x $\in e$ and y $\in e'$}.
- if e is a regular expression, then e* is a regular expression and L(e*) = (L(e))*.
- a regex can use other Boolean operators or the *option* operator such that L(e?) = L(e+ε).

2.2 DFA

A deterministic finite automaton, DFA, is a finite state machine that tests whether a word belongs, or not, to the language described by the regular expression. If in the end of the word the current state is a final state (q is a final state iff q $\in$ F) the word belongs to the language.

A DFA is described as a tuple with:

- Q, a finite set of states.
- Σ, an alphabet, i.e. a finite set of symbols.
- δ: $Q \times A \rightarrow Q$, a transition function, the term deterministic comes from the fact that each transition takes a current state and a symbol has arguments and returns a next state, always and only one for each tuple (state,symbol).
- $q_0 \in Q$, a start state.
- $F \subseteq Q$, a set of final states.

Therefore, we represent the automaton A as a quintuple: A = (Q,Σ,δ,q_0,F).

2.3 NFA

A non-deterministic finite automata (NFA) is an automaton that also describes the acceptance of a language but it has the possibility *to be in several states at once*. This ability is often expressed as an ability to estimate something about its input, for instance, when we use an automaton to search for a certain word in a long text string it can be essential to guess the current position we are in the string that we search using a chain of states to check if the string exists.

Similarly to a DFA, an NFA has a finite set of states, a finite set of symbols *alphabet*, a starting state and a set of accepting states ($final\ states$). They differ on the transition function where the NFA can have δ(X,'a')=Y and δ(X,'a')=Z with $Y \neq Z$, but DFA can not. For the NFA, the transition function takes a state and an input symbol as arguments and can return zero, one or more states, not like the DFA which can only return one, i.e. $\delta \subseteq Q \times A \times Q$.

2.4 Regular Expressions as Trees

In order to test determinism of a regular expression is possible to represent the regex as a tree structure where the nodes are the operators and the leaves are the symbols of the alphabet. To build this tree we have to define:

- $node(regex, path, tree)$, this constructor creates a node in the tree where *regex* is the regular expression denoted in that node, the *path* is a string that represents the path from the root (path = '1') to that node and the *tree* that owns the node.
- $NodeTable$: This table keeps records of a path to each node for an easier access to a node. In the Python implementation, $NodeTable$ represented as a dictionary having paths as keys and nodes as values so that we can get a node from the path. For example, if we are in a concatenation node if we want to get the left child (*Lchild*) we just need to search in our $NodeTable$ for the path of the current node and add a '1' to the path, or a '2' if we want the right child (*Rchild*).

In the tree construction, a node has a left child if is labeled as star, a left and a right child if it is a concatenation or disjunction or no childs if it is a symbol, in this case we add the symbol to our tree's *alphabet* as shown in $Algorithm$ 1. The procedure *Construct* receives a node that expresses an expression and first will build a node for it, then it looks to the expression type and if it is a symbol, then adds it to the language's alphabet, if it is a star expression then he builds it only child (expression without kleene star), otherwise constructs the left child then the right child of the concatenation or the disjunction.

Algorithm 1 Computing a regex tree.

procedure CONSTRUCT(regex,path,t_e)
  $n \leftarrow Node(regex, path, t_e)$
  $NodeTable[path] \leftarrow n$
  if $Atom(n)$ **then**
    $alphabet(t_e) \leftarrow alphabet(t_e) \cup symbol(n)$
  else
    if $lab(n)$ **is** $*$ **then**
      $Lchild(n) \leftarrow Construct(regex.arg, path +' 1', t_e)$
    else
      $Lchild(n) \leftarrow Construct(regex.arg1, path +' 1', t_e)$
      $Rchild(n) \leftarrow Construct(regex.arg2, path +' 2', t_e)$
  return n

2.4.1 LCA

The LCA or lowest common ancestor, is a function that takes two nodes from our structure as arguments and returns the lowest common node in the tree. This can be seen has the smaller sub-expression that contains both sub-expressions (nodes) in the arguments. This function can be implemented in constant time regardless of the length of the expression. [3]

2.4.2 LSA

The LSA, lowest star ancestor, is a function that takes one node from a regex tree as argument and returns the first star-labeled ancestor that the algorithm finds by going through the tree from the argument to the root, if the LSA get to root node and this is not a star-labeled node then the argument has no LSA. [3] With a tree structure organized with sub-expressions in every node where the root node contains the entire regular expression, we can find the LSA of a node in $O(|e|)$ where $|e|$ denotes the length of the expression e.

Chapter 3

Glushkov Automaton

Given the fact that a regular expression defines a pattern of words from a regular language, we can define a set of words that belong to a certain language by neatly browsing the expression that establishes the language. For instance, we can obtain the word $aabaa$ from the expression $a{*}ba{*}$ by using twice the first a, then using b and then, again, twice the second a, resulting on a^2ba^2. It is important to understand that a word can be described by more than one expression and also the other way around. Taking by example the word above, we can also say that $aabaa$ can be accepted by $((aa){*}b{*}){*}$. It is known that, the order of the symbols are crucial in word recognition. From this fact, Victor Glushkov, idealized a construction of a NFA with no ε-transitions. [2]

Glushkov's automata are based on distinguish all symbols from the expression by giving them an index regardless their label, then take them as states and establishing transitions according the follow-up among the symbols.

Below we present the procedure that Glushkov's method implements to mark the regular expression.

$$
\begin{gathered}
mark(\varepsilon, i) = \varepsilon \\
mark(a, i) = a_i \\
mark(a + e, i) = a_i + mark(e, i + 1) \\
mark(a.e, i) = a_i.mark(e, i + 1) \\
mark(e{*}, i) = mark(e, i)
\end{gathered}
$$

The marked symbols are called positions, and the set of positions is represented by $Pos(e)$.

Using one of the above examples, $a * ba*$, we can demonstrate the process by indexing all the symbols, getting (a,1)*(b,2)(a,3)*, and observe that (a,1) is always followed by either (a,1) or (b,2), (b,2) is followed by (a,3) or ∅ and (a,3) by (a,3) or ∅. When a state can be "followed" by ∅ it means that this is a final state. If the word ends there it belongs to the language described by the expression.

Indexing and identifying the symbols is not enough, we need a transition function to determine our automata's edges. So, Glushkov, added three sets to his construction: *First*, *Last* and *Follow*. The $First(e)$ set contains all symbols that can show up in the beginning of the word. The $Last(e)$ set contains all symbols that can appear in the end. The $Follow(e)$ set, is a relation which we represent as a *dictionary* that attributes to every indexed symbol a set of symbols that can follow it. Next, we present how Glushkov's method generates both *First*, *Last* and *Follow* sets.

$$\begin{gathered}
First(\varepsilon) = First(\emptyset) = \emptyset \\
First(\sigma_i) = \sigma_i \\
First(e_1 + e_2) = First(e_1) \cup First(e_2) \\
First(e_1 e_2) = First(e_1) \cup \varepsilon(e_1) First(e_2) \\
First(e*) = First(e)
\end{gathered}$$

$$\begin{gathered}
Last(\varepsilon) = Last(\emptyset) = \emptyset \\
Last(a_i) = a_i \\
Last(e_1 + e_2) = Last(e_1) \cup Last(e_2) \\
Last(e_1 E_2) = Last(e_1) \cup \varepsilon(e_2) Last(e_1) \\
Last(e*) = Last(e)
\end{gathered}$$

$$Follow(\varepsilon, a_i) = Follow(a_i, a_j) = \emptyset$$

$$Follow(e_1 + e_2, a_i) = \begin{cases} Follow(e_1, a_i) & , a_i \in Pos(e_1) \\ Follow(e_2, a_i) & , a_i \in Pos(e_2) \end{cases}$$

$$Follow(e_1 . e_2, a_i) = \begin{cases} Follow(e_1, a_i) & , a_i \in Pos(e_1) \setminus Last(e_1) \\ Follow(e_1, a_i) \cup First(e_2) & , a_i \in Last(e_1) \\ Follow(e_2, a_i) & , a_i \in Pos(e_2) \end{cases}$$

$$Follow(e_1 *, a_i) = \begin{cases} Follow(e_1, a_i) & , a_i \in Pos(e_1) \setminus Last(e_1) \\ Follow(e_1, a_i) \cup First(e_1, a_i) & , a_i \in Pos(e_2) \end{cases}$$

Note that $\varepsilon(e_1)$ implies that e_1 accepts ε and in this case we say that e_1 is nullable.

With these sets we describe Glushkov's automaton as $A\_G = (Pos(e) \cup \{0\}, \Sigma, \delta, Last(e) \cup \varepsilon(0))$, where $\delta(0) = First(e)$ and $\delta(a_i) = Follow(e, a_i)$. The following Lemmas allow the efficient computation of the *Follow* set [3]. We denote by *tree*, the tree representation of the regular expression e.

Lemma 1 *Let* $p, q \in Pos(tree)$, $n = LCA(p, q)$ *and* $s = LSA(p, q)$. *Then* $q \in Follow(p)$ *iff:*

(1) $lab(n) = concat$, $q \in First(Rchild(n))$, $p \in Last(Lchild(n))$.

(2) $q \in First(s)$, $p \in Last(s)$.

We say that $q \in Follow^{\bullet}$(p) iff (1) is satisfied, and $q \in Follow^{*}$(p) iff (2) is satisfied. Note that is possible for some nodes p and q to satisfy both (1) and (2).

Now, we define the Boolean properties *SupFirst* and *SupLast* for every node n, where $n' = parent(n)$:

- $SupFirst(n)$ iff $label(n')$ = concat, n = $Rchild(n')$ and $Lchild(n')$ is not nullable.
- $SupLast(n)$ iff $label(n')$ = concat, n = $Lchild(n')$ and $Rchild(n')$ is not nullable.

With the *Last* set defined is possible to determine the set $FollowAfter$ which indicates possible symbols to appear next to the sub-expression (*node*) we want. To do that we also need a function to determine if a node a is ancestor of a node b, so, we implemented $Reflexive(a, b)$ that takes two arguments and returns true if b is ancestor of a and false otherwise. (see *Algorithm* 2)

Algorithm 2 Computing *Follow* of a sub-expression.

```
procedure followAfter(tree,node)
    S ← ∅
    for p in Last(node) do
        for q in Pos(tree) do
            if Follow(p, q) and ¬Reflexive(q, node) then
                S ∪= q
    return S
```

Chapter 4

Testing Determinism

It was shown before [1] that a regular expression is deterministic iff its Glushkov automaton is deterministic, i.e. a DFA.

In this chapter we describe a determinism test for regular expressions that avoids the construction of Glushkov automaton and can run in linear time.

In *Figure* 4.1 we represent an example of a regex and the corresponding tree annotated with the several properties defined in the previous chapter.

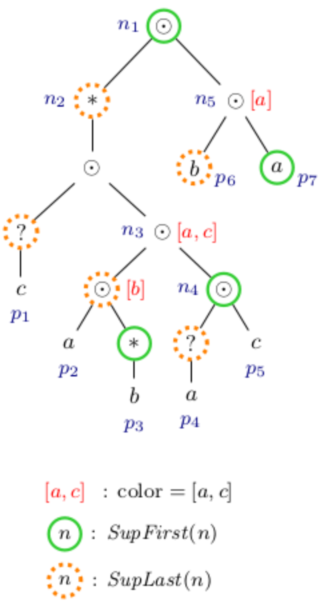

Figure 4.1: Expression e = (c?((ab*)(a?c)))*(ba) represented in a tree. [3]

Let $q, q' \in Nodes(tree)$, where $q \neq q'$ and $n = qSupFirst(q) = qSupFirst(q')$.

With $First$ and $Last$ sets defined for every node in the tree, for $p \in Last(Lchild(n'))$ where n' is the parent of n we can say that $parent(n) = LCA(p, q) = LCA(p, q')$.

By Lemma 1, $q, q' \in Follow(p)$ and by the definition of determinism $lab(q) \neq lab(q')$.

Proposition 1 *$\forall q \neq q' \in Nodes(tree)$, $pSupFirst(q) = pSupFirst(q')$ implies $lab(q)$ $\neq$ lab(q').*

Knowing that a deterministic expression requires a deterministic automaton, if we test every node of the tree, the $Follow$ set cannot have two different atoms with the same color.

Proposition 2 *$\forall a \in \Sigma$ and $n \in Nodes(t_a)$, where t_a is defined below, $Next(n, a)$ contains at most one element.*

The function $Next(n, a)$ returns a set of atoms that belong to the set $Follow(n)$ labeled a.

In order to analyse the regex tree in linear time we establish a color set to each node of our tree meaning that from a position p colored a, the next letter from the word can be a symbol a. So we assign:

- $\forall p \in Pos(tree)$ and $p' = pSupFirst(p)$, include $lab(p)$ in $colors(parent(p'))$.
- p is a *witness* for *color* a in p'.

Note that a node may have more than one color assigned to it, but taking into account *Proposition* 1, each node can have at most one witness per color to be deterministic. This implies that a node n cannot have as witness p_2 and p_4 for color a, this would mean that during the test when we analyse n if we read an a, the automaton would jump to states p_2 and p_4 defining the expression as non-deterministic.

4.1 Skeleton tree

The *skeleton tree* consist on all positions in t_e labeled a, their $pSupLast$ and LSA nodes, and every LCA of two nodes of class a, denoted in t_a.

In our implementation, we defined t_a as a set of pointers to nodes from t_e that also composes the *skeleton tree*. This tree, however, has a peculiarity, not like t_e the *skeleton tree* can have a node labeled concat with only one son, being the second *None*.

In *Figure* 4.2 we can observe that the *a-skeleton* from t_e generates an unambiguous tree that shows the precedence among symbols labeled a.

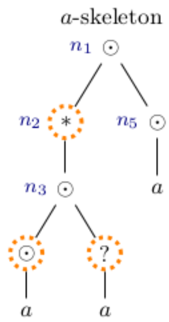

Figure 4.2: Skeleton for color a in the tree of $Figure 1$.[3]

Algorithm 3 Computing the *skeleton tree* for color a.

```
procedure build_t_a(tree,a)
    atoms(t_a) ← ∅
    for n in atoms(tree) do
        if lab(n) = a then
            atoms(t_a) ∪= n
    t_a ← atoms(t_a)
    for x in atoms(t_a) do
        for y in atoms(t_a) do
            if x = y then
                t_a ∪= parent(x, tree)
            else
                t_a ∪= LCA(x, y)
    t_a ∪= find_stars(tree, a)
return t_a
```

The function $find\_stars(tree, a)$ returns a set with all *star*-labeled nodes that are ancestores of any position labeled a from our tree.

4.2 BuildNext

In order to test the determinism of our regex, we implemented a function $BuildNext$ which tests determinism for all nodes in t_e. This function will check every node, one by one, and test if there is at least one node where $Next(n, a)$ has more than one element, or two if $lab(n) = +$. If it occurs in at least one node, than the regex is non-deterministic.

Algorithm 4 [3] Computing the set Next(n,a).

```
procedure BuildNext(a,n,Y)
    if SupLast(n) then
        Y ← ∅
    if n is the left child in t_a of a concat node and
      n has a right sibling n' in t_a and
      (¬SupLast(node) or parent(n, t_a) = parent(n, t_e)) then
        Y ← Y ∪ {FirstPos(n', a)}
    Next(n,a) ← {p ∈ Y | n ≺ p}
    if lab(n) = ∗ then
        Y ← Y ∪ {FirstPos(n, a)}
    if |Y| > 2 then
        return false
    l ← Lchild(n, t_a)
    if l = ∅ then
        return true
    else
        B ← BuildNext(a, l, Y)
    r ← Rchild(n, t_a)
    if r = ∅ then
        return B
    else
        return B ∧ BuildNext(a, r, Y)
```

Executing $BuildNext(n, a, \emptyset)$ the algorithm will add to our tree structure a new dictionary that will link a node n to the *positions* that can come next in the word. This dictionary will be represented as $Next$ and will be denoted as $Next(n)$ to represent $Follow(n)$, or $Next(n, a)$ a subset of $Follow(n)$ that only keeps the positions labeled a. With this we can also realize that $Next(n, a)$ can only have one element, or two if n is labeled +(disjunction).

4.3 CheckNode

Here we define a function that verifies if a sub-expression (represented as a tree node) is deterministic. This function will be later tested for every node in our tree. All nodes (sub-expressions) have to be deterministic so that the expression e is deterministic, in other words $\forall n \in nodes(tree) \land \forall a \in \Sigma\ checknode(n, a)$ returns $true$. This function is defined in $Algorithm$ 5.

Algorithm 5 Verify determinism in node *node*.

```
procedure CHECKNODE(node,sym)
    te ← tree(node)
    ta ← build_t_a(t_e, sym)
    fp ← FirstPos(t_a, n)
    s ← LSA(node)
    r ← Rchild(node)
    if ε(r.exp) then
        if has_next(te, n, sym) then return false
        if s is None then return true
        sl ← pSupLast(node)
        if FirstPos(t_a, s) is fp and Reflexive(s, sl) then return false
    return true
```

With this function we can say that if we apply $checkNode$ for n and his right child is non-nullable then we can say that n is deterministic because there is no possible next symbol in that sub-expression.

If the right child of n is nullable then we need to verify if the sub-expression in n can read the symbol a next, and also if in the path from $LSA(n)$ to $FirstPos(n, a)$ there is nothing non-nullable in the $left$ childs and consequently that $FirstPos(n, a)$ follows the same position p that $Witness(n, a)$ follows.

Algorithm 6 Testing determinism for every node in t_e

procedure CHECKDETERMINISM(tree,n)
  for x **in** $colors(n)$ **do**
    if $\neg checkNode(n, x)$ **then return** false
  if $Lchild(n, tree)$ **is** $None$ **then return** true
  else
    $l \leftarrow checkDeterminism(tree, Lchild(n, tree))$
  if $Rchild(n, tree)$ **is** $None$ **then return** l
  else
    $r \leftarrow checkDeterminism(tree, Rchild(n, tree))$
  return $l \wedge r$

With $checkDeterminism(tree, n)$ (defined in *Algorithm* 6) we first call it with the $root(tree)$ node so that the algorithm tests the function for every node from top to bottom recursively with a depth first search pattern.

This function first calls the function $checkNode$ on n for every color assigned to n, and if any call to the function $checkNode$ returns $false$ than e is non-deterministic, otherwise $checkDeterminism$ will be tested for the childs of n, first to the left like DFS search.

4.4 isDeterministic

In order to verify the determinism of a regex e we test *Proposition* 1 and *Proposition* 2, if any returns $False$ then e is not deterministic. If both return $True$, then we execute $CheckNode(n, a)$ for every $a \in \Sigma$ and every node n coloured a. If any call to the function $CheckNode$ returns $False$ then e is not deterministic.

This test is defined in *Algorithm* 7.

Algorithm 7 Testing determinism for a regex tree.

procedure ISDETERMINISTIC(tree)
  $r \leftarrow root(tree)$
  for x **in** $\Sigma(tree)$ **do**
    $buildNext(x, r, \emptyset)$
  if $\neg$P1$(tree)$ **or** $\neg$P2$(tree)$ **then return** false
  return $checkDeterminism(tree, r)$

Chapter 5

Conclusions

In this project we have revised the Glushkov's automaton construction. Then, we presented a linear time algorithm to test if a regular expression is deterministic using properties acquired from Glushkov's construction, e.g. $First$, $Last$ and $Follow$ sets.

In the appendix we present an implementation of the method studied along with the implementation of a tree structure used in the test for determinism. It could not go without saying that FAdo System made possible the usage of a object oriented structure to assist during the construction of the tree and during the testing by facilitating the access to attributes of each regular expression.

We also performed several experiments of regular expressions generated uniform randomly. In order for this experiences be statistically significant further studies are needed. We also need to implement some of the key functions more carefully in order that the current implementation of the test for deterministic is indeed performed in linear time.

To complete, we note that the linear algorithm for testing determinism extends to regular expressions with numeric occurrence indicators, and that determinism can be decided in linear time.

Appendices

Here we present the code implemented in Python 2.7 within the FAdo system.

```
from os.path import commonprefix
from FAdo.reex import *
from FAdo.cfg import *
from FAdo.fa import *

# Tree Structure
class eTree:
    def __init__(self,regex):
        ReExp = concat(atom("#"),concat(regex.marked(),atom("$")))
        self.NT = dict()
        self.atoms = set()
        self.next = dict()
        self.sub_trees = dict()
        self.alpha = set()
        self.full_tree = construct(ReExp,"1",self)
        self.root = self.NT.get("121")
        self.Color()

    def followList(self):
        l = dict()
        for x in self.atoms:
            f_list = set()
            for w in self.atoms:
                if w.type!=epsilon:
                    if w.follow(x):
                        f_list.add(w)
            l[x]=f_list
        return l

    def build_ta(self,a):
        if self.sub_trees.get(a) is not None:
            return self.sub_trees.get(a)
        ta_atoms = set()
        for n in self.atoms:
            if n.exp.val[0]==a:
                ta_atoms.add(n)
        ta = set()
        for x in ta_atoms:
            for y in ta_atoms:
                if x is y:
                    ta.add(x.parent)
                else:
                    ta.add(self.NT.get(commonprefix([x.path,y.path])))
        self.sub_trees[a] = ta_atoms.union(ta.union(self.find_stars(a)))
        return self.sub_trees.get(a)

    def find_stars(self,a):
```

```
        ta = set()
        for n in self.atoms:
            if n.exp.val[0] == a:
                n = n.parent
                while n != self.root:
                    if n.type is star:
                        ta.add(n)
                    n = n.parent
        return ta

    def has_Next(self,n,a):
        s = self.next.get(n)
        if s is None:
            return False
        for x in s:
            if x.exp.val[0] == a:
                return True
        return False

    def Color(self):
        for x in self.atoms:
            x.pSupFirst().parent.colors.add(x.exp.val[0])
        return

    def isDeterministic(self):
        for x in self.alpha:
            buildNext(x,self.root,set())
        if not self.cond_P1() or not self.cond_P2():
            return False
        return self.check_determinism(self.root)

    def cond_P1(self):
        for a in self.atoms:
            for b in self.atoms:
                if a is b:
                    continue
                if a.pSupFirst() == b.pSupFirst():
                    if a.exp.val[1] == b.exp.val[1]:
                        print "P1"
                        return False
        return True

    def cond_P2(self):
        for a in self.alpha:
            ta = self.build_ta(a)
            for n in ta:
                k = n.FollowAfter
                count = 0
                for x in k:
```

```
                    if x.exp.val[0] == a:
                        count = count + 1
                if count > 1:
                    print "P2"
                    return False
        return True

    def check_determinism(self,n):
        for x in n.colors:
            if not checkNode(n,x):
                return False
        if n.left is not None:
            l = self.check_determinism(n.left)
        else:
            return True
        if n.right is not None:
            return (l and self.check_determinism(n.right))
        return l

# Searches for non-determinism in each node
def checkNode(n,a):
    te = n.tree
    ta = te.build_ta(a)
    f = n.FirstPos(ta)
    s = n.LSA()
    if n.right.exp.ewp():
        if te.has_Next(n,a):
            return False
        if s is None:
            return True
        if s.FirstPos(ta) is f and s.reflexive(n.pSupLas()):
            return False
    return True

# Constructor for eTree structure. Construct nodes.
def construct(reg_exp,path,t):
        node = Node(reg_exp,path,t)
        t.NT[node.path] = node
        if node.type is star:
            node.left = construct(node.exp.arg,path+'1',t)
        elif node.type is concat or node.type is disj:
            node.left = construct(node.exp.arg1,path+'1',t)
            node.right = construct(node.exp.arg2,path+'2',t)
        elif node.type is position:
            sym = node.exp.val[0]
            t.alpha.add(sym)
        return node

class Node:
```

```
    def __init__(self,expr,string,t):
        self.type   = type(expr)
        self.path   = string
        self.exp    = expr
        self.tree   = t
        self.right  = None
        self.left   = None
        self.first  = set()
        self.last   = set()
        self.lsa    = None
        self.supfirst = None
        self.suplast = None
        self.FollowAfter = set()
        self.colors = set()
        if self.type is position:
            t.atoms.add(self)
        if string == '1':
            self.parent = None
        else:
            self.parent = t.NT.get(string[:-1])

    # check if n is ancestor of self
    def reflexive(self,n):
        if n is None:
            return False
        if commonprefix([self.path,n.path])==n.path: #LCA(self,n)
            return True
        return False

    # boolean that states if an atom is the first from the right child of a concatenation
    def SupFirst(self):
        father = self.parent
        if father is not None and father.type is concat:
            if father.right is self and not father.left.exp.ewp():
                return True
        return False

    # boolean that states if an atom is the last from the left child of a concatenation
    def SupLast(self):
        father = self.parent
        if father is not None and father.type is concat:
            if father.left is self and not father.right.exp.ewp():
                return True
        return False

    # pointer to the first(right child) of a concatenation
    def pSupFirst(self):
```

```
        if self.supfirst is not None:
            return self.supfirst
        self.supfirst = None
        if self.SupFirst() or self.path=="1":
            self.supfirst = self
        else:
            self.supfirst = self.parent.pSupFirst()
        return self.supfirst

    # pointer to the last(left child) of a concatenation
    def pSupLast(self):
        if self.suplast is not None:
            return self.suplast
        if self.SupLast() or self.path=="1":
            self.suplast = self
        else:
            self.suplast = self.parent.pSupLast()
        return self.suplast

    #commonprefix is the Python prelude function used for LCA
    def follow(self,p):
        node1 = self.tree.NT.get(commonprefix([p.path,self.path]))
        if node1 is None:
            return False
        if node1.type is concat:
            if self.follow_concat(p,node1):
                return True
        node = node1.LSA()
        if node is not None:
            if self.follow_star(p,node):
                return True
        return False

    # Follow by Concatenation
    def follow_concat(self,p,lca):
        if self in lca.right.First() and p in lca.left.Last():
            return True
        return False

    # Follow by Star
    def follow_star(self,p,lsa):
        if self in lsa.First() and p in lsa.Last():
            return True
        return False

    def First(self):
        if self.first != set():
            return self.first
        for node in self.tree.atoms:
```

```
            if isFirst(self,node):
                self.first.add(node)
        return self.first

    def Last(self):
        if self.last != set():
            return self.last
        for node in self.tree.atoms:
            if isLast(self,node):
                self.last.add(node)
        return self.last

    def LSA(self):
        if self.lsa is not None:
            return self.lsa
        if self.type is star:
            self.lsa = self
        elif len(self.path) == 1:
            self.lsa = None
        else:
            self.lsa = self.parent.LSA()
        return self.lsa

    def FirstPos(self,ta):
        k = set()
        for x in ta:
            if x in self.First():
                k.add(x)
        return k

    def follow_after(self):
        s = set()
        for p in self.Last():
            for q in self.tree.atoms:
                if q.follow(p) and not q.reflexive(self):
                    s.add(q)
        self.FollowAfter = s
        return s

def isFirst(n,p):
    k = p.pSupFirst()
    if k is not None:
        if p.reflexive(n) and n.reflexive(k):
            return True
    return False

def isLast(n,p):
    k = p.pSupLast()
    if k is not None:
```

```
        if p.reflexive(n) and n.reflexive(k):
            return True
    return False

def buildNext(a,n,y):
        if n.SupLast():
            y = set()
        te = n.tree
        ta = n.tree.build_ta(a)
        if ta is None:
            return False
        dad = get_parent(n,ta)
        if dad is not None:
            r = get_right(dad,ta)
            if dad.type is concat and get_left(dad,ta) is n and r is not None:
                if not n.SupLast() or get_parent(n,ta) is n.parent:
                    y = y.union(r.FirstPos(ta))
        n.tree.next[n] = set()
        for x in y:
            if not x.reflexive(n):
                n.tree.next[n].add(x)
        if n.type is star:
            y = y.union(n.FirstPos(ta))
        if len(y)>2:
            return False
        l = get_left(n,ta)
        if l is None or l.exp.ewp():
            return True
        else:
            b = buildNext(a,l,y)
        r = get_right(n,ta)
        if r is None or r.exp.ewp():
            return b
        return (b and (buildNext(a,r,y)))

def get_parent(n,tree):
    t = n.tree
    while n is not t.root:
        parent = n.parent
        if parent in tree:
            return parent
        n = n.parent
    return None

def get_left(n,tree):
    if n.type is position:
        return None
    path = n.path+'1'
```

```
    possible = set()
    for x in tree:
        if len(x.path)>=len(path) and commonprefix([x.path,path])==path:
            possible.add(x.path)
    if len(possible) == 0:
        return None
    return n.tree.NT.get(min(possible))

def get_right(n,tree):
    if n.type is position:
        return None
    path = n.path+'2'
    possible = set()
    for x in tree:
        if len(x.path)>=len(path) and commonprefix([x.path,path])==path:
            possible.add(x.path)
    if len(possible) == 0:
        return None
    return n.tree.NT.get(min(possible))
```

References

[1] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.

[2] Hugo Gouveia. Obtenção de autómatos finítos não determinísticos pequenos. *UC Projeto*, 2009.

[3] Benoît Groz and Sebastian Maneth. Efficient testing and matching of deterministic regular expressions. *J. Comput. Syst. Sci.*, 89:372–399, 2017.